# Techniques of Java Programming: Streams in Java

Manuel Oriol

May 8, 2006

## 1 Introduction

Streams are a way of transferring and filtering information. Streams are directed pipes that transfer information from an input to an output (see Figure 1).

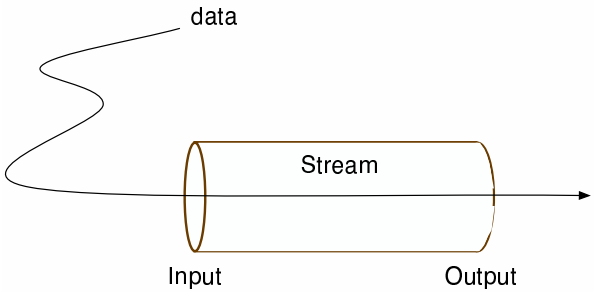


Figure 1: The idea behind streams

Typically, a stream can be built around a device that either receives or flushes information. As a simple example, keyboard interactions behave like that. It is therefore natural to use streams to treat the information flows. Section 2 shows how to instantiate and use streams that flush information (output streams). Section 3 shows how to instantiate and use streams that receive information (input streams). Section 4 shows several examples of commonly used streams.

## 2 Output Streams

In Table 1 we show methods of the abstract class `OutputStream`. The only method to override is the abstract method that writes a byte. Other methods have default implementations that rely on the `write` method and thus do not need to be rewritten.

While `OutputStream` provides the basic facilities for creating streams, it is often easier and more user-friendly to use other classes behaving like an output stream. `PrintStream` is one of them. An example of such a stream is the field `System.out`. `System.out` is a field used to refer to the standard output of the

Table 1: `OutputStream` methods

| **java.io.OutputStream** | |
|---|---|
| void close() | closes the stream |
| void flush() | forces the output stream to proceed the queued elements |
| void write(byte[] b) | writes a byte array in the stream |
| void write(byte[] b, int off, int len) | writes only the bytes between `off` and `off+len` |
| abstract void write(int b) | write a byte |

program. In C for example, this would be referred to as `stdout`. It is also a possible to redefine the standard output by changing this field's binding.[1]

Table 2: `PrintStream` methods

| **java.io.PrintStream** | |
|---|---|
| PrintStream(String fileName) | creates a `PrintStream` outputting to a file. |
| PrintStream(OutputStream out) | creates a `PrintStream` outputing to another stream. |
| void print(Object obj) | prints an Object. This method is overloaded for any type. |
| PrintStream printf(Locale l, String format, Object... args) | sends the formatted arguments in the stream, the format string is similar to C's printf format string. |

The `PrintStream` class (see Table 2) is meant to output objects that it is given in a character-based encoding. The `print` method, applied to regular objects, calls the `toString()` method. If one wants to print objects in a non-standard way one can override this method.

# 3 Input Streams

While `OutputStream`s are meant to provide an easy-to-use object-oriented way for outputing information, `InputStream`s (see Table 3) are meant to provide an easy-to-use object-oriented way for reading information.

The only method to override is the abstract method that reads a byte. Other methods have default implementations that rely on the `read` method and thus do not need to be rewritten.

In order to easily use `InputStream`s it is often useful to define a wrapping `BufferedReader`. As an example:

```
BufferedReader in =
   new BufferedReader(new InputStreamReader(System.in));
```

[1] This can be easily done by using the method `static void setOut(PrintStream out)`

Table 3: `InputStream` methods

| **java.io.InputStream** | |
|---|---|
| int available() | number of bytes available to read. |
| void close() | closes the stream |
| void mark(int readlimit) | marks the current position |
| boolean markSupported() | tests if the stream supports marks |
| abstract int read() | reads the next byte of data |
| int read(byte[] b) | reads as many bytes as `b.length` stores them in the array, returns number of bytes read |
| int read(byte[] b, int off, int len) | read as many bytes as `len` and put them in `b` after `offset` |
| void reset() | puts the stream back to the mark |

In this example, we define a `BufferedReader` around a reader around the `InputStream System.in`. This stream is defined by default on the standard input (from a terminal/command prompt).

Table 4: BufferedReader methods

| **java.io.BufferedReader** | |
|---|---|
| void close() | (as in streams) |
| void mark(int readAheadLimit) | (as in stream) |
| boolean markSupported() | (as in stream returns true) |
| int read() | (as in stream) |
| int read(char[] cbuf, int off, int len) | (as in stream) |
| String readLine() | reads in the stream up to a '\n'. |
| boolean ready() | (as in stream) |
| void reset() | (as in stream) |
| long skip(long n) | skips n characters |

It is then easy to read inputs, line by line, by using `readLine()`. As an example, we can use the NoteTaker application.

```
/**
   * This program takes the first argument as a file name,
   * it opens it and write what user write into it
   */
 public static void main(String[] args){
    String s;

    standard = new BufferedReader(new InputStreamReader(System.in));

    // checks arguments number
    if (args.length!=1) System.exit(0);
```

```
      // open the file name
      try {out = new FileOutputStream(args[0]);}
      catch (FileNotFoundException e){System.exit(0);}

      // users have to leave by using Control-C
      while(true){
         try {
            // read and write
            s=standard.readLine();
            out.write(s.getBytes());
            out.write("\n".getBytes());
         } catch (IOException e){
            System.out.println("I/O error");
            System.exit(0);
         }

      }

   }
```

# 4 Standard Subclasses

Let now see small examples of streams.

## FileInputStream/FileOutputStream

These two types of streams are meant to interact with files. Note that they can be created using the name of the file in the constructor call. Example:

```
PrintStream out = new PrintStream(new FileOutputStream("myfile.txt"));
out.println("My text");
out.close();

BufferedReader reader=new BufferedReader(new FileInputStream("myfile2.txt"));

// this time we append
out=new PrintStream(new FileOutputStream("myfile.txt"), true);
out.print("\t"+in.readLine());
```

## ObjectInputStream/ObjectOutputStream

These streams are meant to easily dump objects into byte arrays - `byte[]`. Note that if a class needs its objects to be serialized, it needs to implement `Serializable`. As an example, to save and read an object into and from a file:

```
ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("myfile.txt"));
out.writeObject("test");
out.close();
...
```

```
ObjectInputStream ois=new ObjectInputStream(new FileInputStream("myfile.txt"));
String s;
// this time we read the object
s=(String)ois.readObject();
```

### SocketInputStream/SocketOutputStream

These streams are used to communicate through sockets.

```
Socket s;
...
PrintStream out = new PrintStream(s.getOutputStream());
out.print("EOF");
...
BufferedReader reader=new BufferedReader(s.getInputStream());

// this time we read a line
String s=reader.readLine();
```